A Relational Interpreter for Synthesizing JavaScript

ARTEM CHIRKOV, University of Toronto Mississauga, Canada
GREGORY ROSENBLATT, University of Alabama at Birmingham, USA
MATTHEW MIGHT, University of Alabama at Birmingham, USA
LISA ZHANG, University of Toronto Mississauga, Canada

We introduce a miniKanren relational interpreter for a subset of JavaScript, capable of synthesizing imperative, S-expression JavaScript code to solve small problems that even human programmers might find tricky. We write a relational parser that parses S-expression JavaScript to an intermediate language called LambdaJS, and a relational interpreter for LambdaJS. We show that program synthesis is feasible through the composition of these two disjoint relations for parsing and evaluation. Finally, we discuss three somewhat surprising performance characteristics of composing these two relations.

CCS Concepts: • **Software and its engineering → Functional languages**; **Constraint and logic languages**.

Additional Key Words and Phrases: miniKanren, logic programming, relational programming, program synthesis, JavaScript

1 INTRODUCTION

Program synthesis is an important problem, where even a partial solution can change how programmers interact with machines [7]. The miniKanren community has been interested in program synthesis for many years, and has used relational interpreters to solve synthesis tasks [5, 6, 10, 13].

JavaScript is famously unpredictable [3]. However, JavaScript's ubiquity makes it an important language to target for program synthesis. To that end, this paper explores the feasibility of both writing a relational JavaScript interpreter, and using such an interpreter to solve small synthesis problems. To remind readers of the quirkiness of JavaScript, we invite you to try the puzzle in Figure 1, inspired by a similar example in Guha et al. [8].

```
(function(x) {
  return (function() {
    if (false) {
      ____________________
    }
    return x;})();
})(42);
```

Fig. 1. A JavaScript puzzle: fill in the blank so that the entire expression evaluates to 'undefined' rather than '42'.

Authors' addresses: Artem Chirkov, University of Toronto Mississauga, Canada, artem.chirkov@mail.utoronto.ca; Gregory Rosenblatt, University of Alabama at Birmingham, USA, gregr@uab.edu; Matthew Might, University of Alabama at Birmingham, USA, might@uab.edu; Lisa Zhang, University of Toronto Mississauga, Canada, lczhang@cs.toronto.edu.

miniKanren.org/workshop/2020/8-ART2

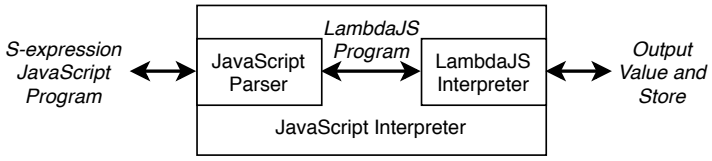

Fig. 2. Relational JavaScript Interpreter Design

Our main contribution is a proof-of-concept relational interpreter for a subset of JavaScript[1]. We build our JavaScript interpreter by combining two components:

(1) A relational interpreter for an intermediate language called LambdaJS [8]. LambdaJS describes the core semantics of JavaScript. We describe this interpreter in Section 2.
(2) A relational parser that desugars an S-expression representation of a JavaScript program into a LambdaJS program. We describe the S-expression representation and the desugaring process in Section 3.

The LambdaJS interpreter is capable of synthesizing LambdaJS code, but in order to synthesize actual S-expression JavaScript, we need to also use the relational parser from JavaScript to LambdaJS. When we combine the two relations, we have a relational JavaScript interpreter capable of synthesizing S-expression JavaScript code to solve synthesis problems like the one in Figure 1.

In addition to presenting synthesis experiments, we describe three performance characteristics of composing the parser and interpreter relations that are somewhat surprising. These observations may be helpful for miniKanren users even if they are not interested in JavaScript.

2 A RELATIONAL LAMBDAJS INTERPRETER

JavaScript syntax is complex, and its semantics is quirky and inconsistently defined [3]. Instead of building a relational interpreter directly, we first translate JavaScript syntax to a simpler, intermediate language called LambdaJS [8]. LambdaJS was initially developed in 2010 based on ECMAScript 3. Although the ECMAScript standards are regularly updated, we work with an older subset of JavaScript and we do not support the latest features.

2.1 LambdaJS

LambdaJS is a simple language that expresses the fundamental features of JavaScript. We can desugar any JavaScript program into a LambdaJS program. LambdaJS consists of these core language features:

- atomic values (number, string, boolean, undefined, null)
- immutable, statically-scoped variables, let expressions and functions
- mutable references (allocate, dereference, assign, delete)
- objects with prototype inheritance
- basic control flow constructs (begin, if, while)
- control effects (throw, break, try, catch, finally, label)

Other JavaScript syntax, such as `for` loops and `switch` statements, can be desugared into LambdaJS. Interested readers can find the details of the LambdaJS semantics in Guha et al. [8]. LambdaJS programs tend to be verbose, and are not meant to be written by hand. Instead, LambdaJS is intended to simplify tasks related to reasoning about the semantics of JavaScript, such as writing an interpreter.

[1] https://github.com/Artish357/RelateJS

2.2 LambdaJS Interpreter Implementation

Our implementation of a LambdaJS interpreter roughly follows Guha et al. [8], which defines a small-step operational semantics for LambdaJS. We implement a big-step interpreter consistent with these semantics. The rest of this section describes our implementation choices and details, and notes where our implementation differs from Guha et al. [8] .

2.2.1 Atomic values. Our implementation of LambdaJS includes these atomic types: `null`, `undefined`, `number`, `string`, and `boolean`. To differentiate these types, we use tagging: every value is represented by a list, with the first element being a symbol describing its type. For simplicity, we restrict numbers to only natural numbers, and represent them using the relational little-endian numerals from Kiselyov et al. [9]. We represent a string as a list of such numbers, where each number encodes a character.

2.2.2 Immutable variables, let expressions, and functions. LambdaJS is statically scoped, and variables are immutable. Once bound, a variable is never re-bound, but may be shadowed by a new binding. In our implementation, these variable bindings are stored in an immutable environment represented by an association list (list of key-value pairs). New variable bindings are created when functions are called and by `let` expressions.

2.2.3 Mutable References. Like JavaScript, LambdaJS is an imperative language that allows mutation. However, LambdaJS makes mutable references explicit, and assigns these references in a separate *store*. The store contains a list of values that are accessed by position, where a position is represented as a list-encoded Peano numeral. Our LambdaJS interpreter implements the following memory operations:

- `allocate`: put a value into the store at a new position, and increment the position counter.
- `dereference`: retrieve a value from the store.
- `assign`: put a value into the store at an existing position.

Memory operations are side-effects that are threaded through the evaluation. Subsequent evaluations can see the effects of previous evaluations. In particular, the interpreter has four additional parameters that do not appear in relational interpreters for functional languages: an initial value for the mutable store, a final value for the mutable store, an initial value for the mutable store index counter, a final value for the mutable store index counter.

2.2.4 Functions and Objects. LambdaJS functions are distinct from JavaScript functions. In JavaScript, a function is an object, so that statements like these are valid: `(function(x){ return x; })["why"] = 42`. In contrast, LambdaJS functions are simpler, and evaluate to a closure containing the environment at the time the function was defined.

LambdaJS objects are also distinct from JavaScript objects. In JavaScript, we manipulate objects indirectly through memory references (explained in detail in Section 3.2.2). LambdaJS objects contain accessible fields, which we implement using an association list. LambdaJS objects are immutable, and programs can manipulate objects functionally using the following operations:

- `get`: retrieve the value of an object field;
- `create/update`: create an object field and assign its value, or update the field if it already exists; return a new object with the new/updated fields;
- `delete`: remove an object field, and return a new object with the field removed.

The LambdaJS object semantics described in Guha et al. [8] include prototype inheritance. However, prototype inheritance can instead be implemented by the desugaring stage. For simplicity, our proof-of-concept interpreter does not support prototype inheritance.

2.2.5 Basic Control Flow. Our implementation of LambdaJS supports three basic control flow operations:

- `begin`: A `begin` expression sequences two subexpressions. Its value is the value of the second subexpression, although the first could have side-effects (alter the mutable store, throw an exception).
- `if`: An `if` expression consists of a condition, a `then` branch, and an `else` branch.
- `while`: A `while` loop expression consists of a condition and a body. A terminating `while` loop always eventually evaluates to `undefined`, although the body expression could have side-effects.

Other JavaScript control flow operations, such as `for` loops, can be expressed in terms of these operations.

2.2.6 Handling control effects. A control effect stops normal control flow, and jumps to an exception handler (if one exists). Control effects include `throw`, `break` and `return`. Our handling of control flow deviates slightly from Guha et al. [8]: we consolidate `throw` and `break` into a single `break` effect, and we combine `try`, `catch`, `finally`, and `labels` into a combined construct that catches a `break` with a particular label, binds the value carried by the `break` to a variable, and evaluates a handler expression.

In order to properly handle exceptions, we need a way to interrupt normal control flow. We check the result of an evaluation for a `break` value. If there is a `break` value, we stop the computation and return to the correct exception handler (if one exists). Otherwise, we continue normally. In our implementation, we model this behaviour using a macro called `effect-propagateo`.

3 A RELATIONAL JAVASCRIPT TO LAMBDAJS PARSER

Our parser desugars an S-expression JavaScript statement into a LambdaJS expression. The parser needs to be relational so that we can use the desugarer as a "sugarer" that generates S-expression JavaScript from the LambdaJS equivalent.

Desugaring or parsing is a non-trivial process since much of the complexity of JavaScript is encoded in the parser. For example, JavaScript is a statement-based language, and LambdaJS is an expression-based language. JavaScript functions are objects, whereas LambdaJS functions are not. JavaScript variables and objects have implicit mutable reference semantics, and LambdaJS makes mutable references and their operations explicit. JavaScript also has richer syntax for control flow operators, some of which we support in our work.

In Section 3.1, we describe the S-expression syntax that we use to represent a subset of JavaScript. Then, in Section 3.2, we describe the design choices we made to transform S-expression JavaScript into LambdaJS.

3.1 S-expression JavaScript syntax

The grammar for the S-expression JavaScript syntax is shown in Figure 3. In particular, the subset of JavaScript that we support includes:

- atomic values (same as in Section 2.2.1)
- variable allocation/assignment/access
- object creation, object field access/update
- function definition, function call
- if statements
- for/while loops
- try/catch/finally

Additionally, Figure 4 contains a side-by-side comparison of JavaScript syntax, and the corresponding S-expression notation.

Since relational arithmetic numbers and strings are difficult for humans to read, we add a layer to our parser that transforms human-readable literals into the relational number and string format described in Section 2.2.1.

```
Symbol         = ... infinite set of symbols ...
Number         = ... infinite set of numbers ...
String         = ... infinite set of strings ...
Operation      = + | - | * | / | < | string+ | string< | char->nat | nat->char
VarDeclaration = Symbol | (Symbol E)
L              = Number | String | #t | #f | (undefined) | (null)
E              = L | Symbol | (object (String E) ...) | (op Operation E ...)
               | (@ E E) | (:= Symbol E) | (:= (@ E E) E) | (call E E ...)
               | (function (Symbol ...) E ...)
S              = E | (var VarDeclaration ...) | (begin S ...) | (if E S S)
               | (for (S E E) S ...) | (return E) | (throw E) | (break)
               | (try S catch Symbol S) | (try S finally S)
               | (try S catch Symbol S finally S) | (while E S ...)
```

Fig. 3. S-expression JavaScript grammar

3.2 Relational Parser Implementation

The relational parser desugars an S-expression JavaScript statement into a LambdaJS expression. We make a distinction between JavaScript *statements* and *expressions*. At the top level, our parser expects a JavaScript program to be a single statement.

The rest of this section describes choices we made in converting a S-expression JavaScript statement into a LambdaJS expression.

3.2.1 Variables. Unlike in LambdaJS, JavaScript variables are mutable. Therefore, the parser desugars a JavaScript variable declaration by creating a LambdaJS immutable variable bound to an allocated mutable reference. By assigning the value of the underlying reference, we can express mutation through immutable variables. In order to access the values in these underlying references, the parser dereferences all variables before use.

The parser also implements JavaScript hoisting (or lifting) semantics, where variable declarations are moved to the top of their scope. This implies that variables defined using `var` are visible even before reaching the declaration location. Hoisting is performed in function definitions, and at the top level.

3.2.2 Objects. A JavaScript object does not translate directly into a single LambdaJS object. Instead, it is represented as a LambdaJS object containing two more objects stored in the fields "public" and "private". When the JavaScript program accesses or updates an object field, we use the corresponding fields in the "public" LambdaJS object. The "private" object contains hidden fields used to recognize and call functions. (See Section 3.2.3.)

For simplicity, we don't implement JavaScript object prototype inheritance. However, this feature can be implemented in a similar way using the "private" object.

3.2.3 Function Definitions. In JavaScript, a function is represented as an object with a special private "call" field. The standard object functionalities, such as field access and assignment, are available for functions.

For simplicity, we don't implement the implicit `this` variable, but we could do so by extending the environment when a function is called.

3.2.4 Function Calls. Since JavaScript functions are objects, during a function call we access the private "call" field of the object to obtain the corresponding LambdaJS function. We apply arguments in the usual way, but

```
// JavaScript Variables
var x;
var y = 3;
var a = y, b = "hi", f;
y;
```

```
;; Corresponding S-expression Notation
(var x)                    ; allocation
(var (y 3))                ; declaration
(var (a y) (b "hi") f)     ; many assignments
y                          ; use
```

```
// JavaScript Objects
{"a": 2, "b": false};
x["a"];
x["a"] = null;
```

```
(object ("a" 2) ("b" #f))  ; object literal
(@ x "a")                  ; field access
(:= (@ x "a") (null))      ; field update
```

```
// JavaScript Functions and Built-in
y = y + 1;
f = function(m, n) {
  return false;
};
f(x y);
f["t"] = 3;
```

```
(:= y (op + y 1))          ; built-in operations
(:= f (function (m n)      ; function definition
    (return #f)))

(call f x y)               ; function call
(:= (@ f "t") 3)           ; functions are objects
```

```
// JavaScript if statements
if (y === undefined) {
    x["a"] = 3;
} else {
    x["a"] = 2;
}
```

```
(if (op === y (undefined)) ; condition
    (:= (@ x "a") 3)       ; then-branch
    (:= (@ x "a") 2))      ; else-branch
```

```
// For Loops
for (var i = 0;
     i < 10;
     i = i + 1) {
    f(x, i);
    break;
}
```

```
(for ((var (i 0))          ; initializer
      (op < i 10)          ; condition
      (:= i (op + i 1)))   ; update
   (call f x i)            ; body
   (break))
```

```
// while loops
while (i < 10) {
    i = i + 1;
}
```

```
(while (op < i 10)         ; condition
   (:= i (op + i 1)))      ; body
```

Fig. 4. A sample of our S-expression JavaScript syntax

before evaluating the body of the function, we re-bind the function parameters to mutable references containing the values of the incoming arguments.

3.2.5 Built-in Operations. For simplicity, the syntax for calling built-in operations differs from that of calling functions. The JavaScript built-in operations we support translate directly to LambdaJS built-in operations. We

support numeric operations (`+`, `-`, `*`, `/`, `<`) string operations (`string+`, `string<`, `char->nat`, `nat->char`), `typeof` and strict equality `===`.

3.2.6 Assignments. The semantics of the JavaScript assignment operator differs depending on whether we are assigning to a variable or to an object field. A JavaScript variable assignment corresponds to assigning its LambdaJS mutable reference to the value on the right hand side.

A JavaScript object field assignment (like `x["a"] = 2`) is a bit more complicated. We expect the object expression (in this case `x`) to evaluate to an object reference. We dereference it to obtain the "public" LambdaJS object, then use the `create/update` operation to update the object's field. The field update produces a new LambdaJS object, so we assign the reference to the new object.

3.2.7 Control Flow. We support JavaScript `if` statements, `while` and `for` loops, and a `begin` statement. Unlike in LambdaJS, these operations are *statements* rather than expressions.

An S-expression JavaScript `begin` statement contains a list of statements to be executed sequentially. It is equivalent to the semantics of the semicolon `;` in pure JavaScript.

Both `if` and `while` statements map almost directly to their LambdaJS counterparts. However, their bodies are statements and not expressions. The JavaScript `for` loop syntax is converted into a LambdaJS `while` loop in the typical way.

3.2.8 Control Effects. Effects like JavaScript `return` and `break` are translated into a LambdaJS `throw` that are caught by a LambdaJS `catch` with a hard-coded label (with the label name "return" or "break"). When functions are defined, their bodies are wrapped in a `catch` for the "return" label. Loop bodies are wrapped in a `catch` for the "break" label. The catch handler for the "return" evaluates to the thrown value. The catch handler for the "break" ignores the thrown value.

JavaScript `throw` is also translated to a LambdaJS `throw`. It is caught by a LambdaJS `catch` with a hard-coded label (with the label name "error"). A LambdaJS `catch` is produced when translating JavaScript `try/catch`, `try/catch/finally` and `try/finally`. Additionally, for JavaScript `try/catch` and `try/catch/finally`, the LambdaJS `catch` handler binds a variable to the thrown value.

4 JAVASCRIPT SYNTHESIS EXPERIMENTS

We compose the parser and the interpreter so that we can run the combined relation "backwards" to synthesize S-expression JavaScript. All of our experiments are performed on a Macbook Pro with a 2.7 GHz Intel Core i5 processor and 16 GB RAM.

4.1 The Puzzle in Figure 1

We use our interpreter to solve the puzzle in Figure 1. We translate the JavaScript puzzle into S-expression syntax, and use a logic variable to represent the blank in the then-branch of the `if` statement. In our S-expression syntax an `if` statement always requires a then-branch and an else-branch, so we add a trivial else-branch.

```
(run 1 (BLANK)
  (fresh (code store)
    (parseo/readable
      `(call (function (x)
               (return (call (function ()
                               (if #f      ; condition
                                   ,BLANK  ; then-branch
                                   #f)     ; else-branch
                               (return x)))))
             42)
      code)
    (evalo code (jundef) store)))
```

Running the above query, we obtain the following answer:

```
(var (x _.0))  ; where _.0 is any symbol
```

This answer is equivalent to the JavaScript code:

```
var x = <symbol>; // where <symbol> is any variable name
```

This answer leverages *hoisting*, the mechanism where variables and function declarations are moved to the top of their scope (Section 3.2.1). Even though this declaration of the variable `x` is unreachable, it is still moved to the top of the inner function scope.

If we modify the query to return multiple answers, we get similar answers:

```
((var (x _.0)) (sym _.0))  ;; var x = <symbol>;
(var (x #t))               ;; var x = true;
(var (x #f))               ;; var x = false;
(var x)                    ;; var x;
...
```

These answers translate to the following answers to the puzzle in Figure 1:

```
(function(x) {
   return (function() {
     if (false) {
       var x; // OR var x = <value>, OR EVEN var x = x;
     }
     return x;})();
})(42);
```

4.2 Synthesize Input Data

In this section we use our interpreter to generate numeric values that satisfy some simple constraints written in JavaScript. We will use the imperative `modulo` function defined below to express modular arithmetic constraints.

```
var modulo = function(n, m) {
  while (< m n) {
     n = n - m;
  }
  return n;
};
```

We can run this program "backwards" to find, for example, inputs `n` where `modulo(n, 3) == 1`. Running such a test yields the answers `(1 4 7 10 13 19 31 37 22 43 ...)`. Notice that the answers are not sorted: an artifact of the relational arithmetic operation.

If we add an additional constraint, so that we look for values `n` where `modulo(n, 3) == 1` and `modulo(n, 5) == 2`, we get these 10 answers in 24 seconds:

```
(7
 37
 22
 (op char->nat "\a")      ;  7
 (op char->nat "%")       ; 37
 (op char->nat "\u0016")  ; 22
 67
 52
 (op char->nat "C")       ; 67
 127
 ...)
```

The results contain not only numeric literals, but also calls to character-to-numeric conversion operations! The `char->nat` calls evaluate to the same values as the integer literals, and appear in the same order. As is usual for miniKanren relational interpreters, it becomes progressively slower to generate more and more answers: generating 20 answers takes 146 seconds.

4.3 Synthesizing Imperative Code

We use the interpreter to synthesize the loop body of the definition of `sum_of_range` function below.

```
var sum_of_range = function(n) {
  var total = 0;
  for (var i = 0; i < n; i = i + 1) {
    ____________________
  }
  return total;
};
```

We provide the two examples: `sum_of_range(3) = 3` and `sum_of_range(4) = 6`. Using faster-miniKanren [2], our relational interpreter synthesizes the update rule `var total = total + i;` in about 5 minutes. It might be surprising that the synthesized code uses an unnecessary `var` to perform the assignment, rather than using a simple assignment operation. This behaviour is an artifact of the search order of the parser.

We also try to synthesize the underlined portions of the `sum_of_range` function. We use the same two examples, and provide the rest of the function while replacing one of the underlined code fragments with a logic variable.

```
var sum_of_range = function(n) {
  var total = 0(1);
  for (var i = 0(2); i < n(3); i = i + 1(4)) {
    total = total + i
  }
  return total;
};
```

Using faster-miniKanren [2], our relational interpreter synthesizes:

(1) the entire declaration `var total = 0` in 0.4 seconds;
(2) the entire declaration `var i = 0` in 0.2 seconds;
(3) the stopping condition `i < n` in 0.6 seconds.

The loop increment `i = i + 1` did not synthesize within 10 minutes. This result is not surprising, given how unconstrained the control flow is without the loop increment information.

4.4 Synthesizing Recursive Code

To see how well our interpreter can reason about recursive code, we attempt to synthesize each of the underlined portions of a recursive definition of the `fibonacci` function.

```
var fibonacci = function(n) {
  if (n < 2(1)) {
    return n(2);
  } else {
    return fibonacci(n - 1(3)) + fibonacci(n - 2(4));
  }
};
```

We provide the following two examples: `fibonacci(2) = 1` and `fibonacci(5) = 5`. Using faster-miniKanren [2], our relational interpreter synthesizes the underlined portions of the code:

(1) the condition `n < 2` in 32 seconds;
(2) the base case `return n;` in 0.3 seconds;
(3) the argument `n - 1` to the first recursive call in 68 seconds;
(4) the argument `n - 2` to the second recursive call in 10 seconds.

The difference in run time in the last two experiments is due to the order of evaluation. When solving for the first recursive call, the value of the second recursive call is not yet computed. In contrast, when solving for the second recursive call, we have already computed the first.

4.5 Synthesizing LambdaJS Code Directly

Surprisingly, direct synthesis of LambdaJS through the interpreter (without the JavaScript parser) is sometimes slower than synthesizing the equivalent JavaScript code.

To demonstrate, we use the relational LambdaJS interpreter to solve the same problems as in Section 4.3 and Section 4.4. We also use the same examples for each synthesis problem, but encode each problem as a LambdaJS program rather than a JavaScript program.

Table 1 shows the synthesis time comparison for each problem. Most of the LambdaJS synthesis tasks do not succeed within 10 minutes, and the two that do so only succeed because we simplify the tasks. These simplifications are necessary because a JavaScript `var` declaration desugars into two LambdaJS operations: a mutable reference allocation and an assignment at a different program location. Synthesizing the full variable initializations requires synthesizing both components, which takes more than 10 minutes. Instead, the table reports the time to synthesize just the assignment.

Table 1. Comparison of JavaScript vs LambdaJS Synthesis Time

Problem	**Blank**		**JavaScript**	**LambdaJS**
`sum_of_range`	Initialization	`var total = 0`	0.4 seconds	*5 seconds
`sum_of_range`	Loop initialization	`var i = 0`	0.2 seconds	*0.2 seconds
`sum_of_range`	Loop stop condition	`i < n`	0.6 seconds	(>10 min)
`sum_of_range`	Loop increment	`i = i + 1`	(>10 min)	(>10 min)
`sum_of_range`	Loop body	`total = total + i`	5 minutes	(>10 min)
`fibonacci`	Base case condition	`n < 2`	32 seconds	(>10 min)
`fibonacci`	Base case body	`return n;`	0.3 seconds	(>10 min)
`fibonacci`	First recursive call	`n - 1`	68 seconds	(>10 min)
`fibonacci`	Second recursive call	`n - 2`	10 seconds	(>10 min)

5 DISCUSSION

The parser-interpreter design of our relational JavaScript interpreter has both advantages and disadvantages. Some of these are straightforward: separating the relations makes the code readable, and combining (or interleaving) the relations would likely make synthesis faster. Through this work, we found a few characteristics of the parser-interpreter design that are not immediately obvious.

(1) **The parser almost entirely constrains the program structure.** The results in Section 4.5 show that synthesizing LambdaJS without the parser is *slower* than synthesizing S-expression JavaScript. This behaviour sounds paradoxical, but really isn't. Not all LambdaJS programs have a JavaScript equivalent, so the parser greatly reduces the search space and improves synthesis performance.

(2) **The ordering of conde branches in the LambdaJS interpreter does not affect JavaScript synthesis performance.** Typically, rearranging the `conde` branches of a relation alters its performance characteristics. However, since the parser almost entirely constrains the program structure, there are few decisions to be made by the LambdaJS interpreter. Rearranging the `conde` branches in the parser *does* change the performance characteristics, as one would expect.

(3) **The parser-interpreter combination is not refutationally complete.** In other words, a search for a nonexistent solution will generally not terminate. The constraint on the output LambdaJS program is only seen after we begin interpreting that LambdaJS program, which only happens after leaving the call to the parser. The parser does not get feedback to constrain its space of possible JavaScript programs. Therefore, there is an infinite number of JavaScript programs to enumerate.

6 RELATED WORK

The idea of a relational interpreter written in miniKanren is not new. Byrd et al. [6] implements a small Scheme interpreter capable of synthesizing quines. Byrd et al. [5] extends and optimizes the interpreter. Though published relational interpreters written in miniKanren tend to be for functional languages, Lundquist et al. [10] implemented a relational interpreter that can synthesize x86 assembly.

Beyond the miniKanren community, many researchers are interested in synthesizing imperative programs. Solar-Lezama [12] allows a programmer to specify an imperative program containing holes where expressions should be synthesized, similar to how we can specify a program containing logic variables. Their solving technique focuses on imperative programs, but includes a limited ability to reason about recursive computations by inlining them a fixed number of times. Blanc et al. [4] similarly allows a programmer to specify a program containing holes, supporting synthesis of a subset of Scala including support for mutation and loops. Their solving technique is able to reason about arbitrary recursive computations. Both Solar-Lezama [12] and Blanc et al. [4] restrict holes to stand for proper subexpressions. This choice makes the synthesis problem more tractable than allowing holes to be placed anywhere, and these tools achieve competitive benchmark performance. In contrast, our use of miniKanren gives us the freedom to place logic variables anywhere in a program, allowing us to synthesize any syntactic fragment, but makes it more difficult to achieve competitive performance.

There has been little published work that focuses exclusively on synthesizing JavaScript, possibly due to JavaScript's complexity. Some works apply a general purpose technique to JavaScript. For example, Padhye et al. [11] applies their semantic fuzzing technique to generate syntactically correct JavaScript for the purpose of finding flaws in JavaScript implementations. However, their application does not require reasoning about the semantics of the generated code. Other work on JavaScript synthesis focuses on narrow subsets of the language relevant to web programming, such as Bajaj et al. [1], which synthesizes DOM selectors by example.

7 CONCLUSION

We described a proof-of-concept relational interpreter for a subset of the JavaScript programming language, formed by composing a JavaScript parser and LambdaJS interpreter implemented as two separate relations. We have demonstrated that this interpreter can be used to solve small, and sometimes tricky, synthesis problems. We also described some characteristics of the parser-interpreter design that were not obvious.

ACKNOWLEDGMENTS

We thank William E. Byrd and the anonymous reviewers for their helpful comments. We also thank Ina Jacobson for helpful feedback on the writing. Research reported in this publication was supported in part by the National Center For Advancing Translational Sciences of the National Institutes of Health under Award Number OT2TR003435. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

REFERENCES

[1] Kartik Bajaj, Karthik Pattabiraman, and Ali Mesbah. 2015. Synthesizing Web Element Locators. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 331–341.

[2] Michael Ballantyne and William E. Byrd. 2015. A fast implementation of miniKanren with disequality and absento, compatible with Racket and Chez. https://github.com/michaelballantyne/faster-miniKanren

[3] Gary Bernhardt. 2012. Wat. A lightning talk from CodeMash, 2012. https://www.destroyallsoftware.com/talks/wat

[4] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. 2013. An overview of the Leon verification system: Verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala*. 1–10.

[5] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP (2017), 8.

[6] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, live and untagged: Quine generation via relational interpreters (programming pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming*. ACM, 8–29.

[7] Molly Q Feldman, Yiting Wang, William E. Byrd, François Guimbretière, and Erik Andersen. 2019. Towards answering "Am I on the right track?" automatically using program synthesis. In *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. 13–24.

[8] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. The essence of JavaScript. In *European conference on Object-oriented programming*. Springer, 126–150.

[9] Oleg Kiselyov, William E. Byrd, Daniel P Friedman, and Chung-chieh Shan. 2008. Pure, declarative, and constructive arithmetic relations (declarative pearl). In *International Symposium on Functional and Logic Programming*. Springer, 64–80.

[10] Gilmore R. Lundquist, Utsav Bhatt, and Kevin W. Hamlen. 2019. Relational Processing for Fun and Diversity. In *Proceedings of the 2019 miniKanren and Relational Programming Workshop. Harvard Computer Science Group Technical Report TR-02-19*. 100–113.

[11] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340.

[12] Armando Solar-Lezama. 2008. *Program synthesis by sketching*. Ph.D. Dissertation. PhD thesis, EECS Department, University of California, Berkeley.

[13] Lisa Zhang, Gregory Rosenblatt, Ethan Fetaya, Renjie Liao, William E. Byrd, Matthew Might, Raquel Urtasun, and Richard Zemel. 2018. Neural guided constraint logic programming for program synthesis. In *Advances in Neural Information Processing Systems*. 1737–1746.

A RELATIONAL LAMBDAJS INTERPRETER

```
;; Entry point for the LambdaJS interpreter
(define (evalo expr val store~)
  (fresh (next-address~)
    (eval-envo expr '() val '() store~ '() next-address~)))

;; LambdaJS interpreter with store
(define (eval-envo expr env value
                   store store~                ; mutable store before/after
                   next-address next-address~) ; index counter for the store
  (conde
    ;; Atomic values (Section 2.2.1)
    ((== expr value)
     (== store store~)
     (== next-address next-address~)
     (conde ((fresh (payload) (== expr (jrawnum payload))))
            ((fresh (bindings) (== expr (jobj bindings))))
            ((fresh (b) (== expr (jbool b))))
            ((fresh (payload) (== expr (jrawstr payload))))
            ((== expr (jundef)))
            ((== expr (jnul)))))
    ;; Immutable variable lookup (Section 2.2.2)
    ((fresh (var) ;; Look up a variable
       (== expr (jvar var))
       (== store store~)
       (== next-address next-address~)
       (lookupo var env value)))
    ;; Builtin operations (Section 3.2.5)
    ((fresh (rator rands args value^)
       (== expr (jdelta rator rands))
       (eval-env-listo rands env value^ store store~ next-address next-address~)
       (effect-propagateo value^ value store~ store~ next-address~ next-address~
         (== value^ (value-list args))
         (conde
           ((fresh (v1 v2 digits1 digits2 result remainder) ; numeric operations
              (== `(,v1 ,v2) args)
              (typeofo v1 (jstr "number") store~)
              (typeofo v2 (jstr "number") store~)
              (== `(,(jrawnum digits1) ,(jrawnum digits2)) args)
              (conde ((== rator '+)
                      (== value (jrawnum result))
                      (pluso digits1 digits2 result))
                     ((== rator '-)
                      (== value (jrawnum result))
                      (minuso digits1 digits2 result))
```

```
                  ((== rator '*)
                   (== value (jrawnum result))
                   (*o digits1 digits2 result))
                  ((== rator '/)
                   (== value (jrawnum result))
                   (/o digits1 digits2 result remainder))
                  ((== rator '<)
                   (conde ((== value (jbool #t)) (<o digits1 digits2))
                          ((== value (jbool #f)) (<=o digits2 digits1)))))))
       ((fresh (v1 v2) ; ===
          (== `(,rator ,args) `(=== (,v1 ,v2)))
          (conde ((== value (jbool #t)) (== v1 v2))
                 ((== value (jbool #f)) (=/= v1 v2)))))
       ((fresh (v1) ; typeof
          (== `(,rator (,v1)) `(typeof ,args))
          (typeofo v1 value store~)))
       ((fresh (str char) ; char->nat
          (== `(,str) args)
          (typeofo str (jstr "string") store~)
          (== `(,(jrawstr `(,char))) args)
          (== rator 'char->nat)
          (== value (jrawnum char))))
       ((fresh (num digits) ; nat->char
          (== `(,num) args)
          (typeofo num (jstr "number") store~)
          (== `(,(jrawnum digits)) args)
          (== rator 'nat->char)
          (== value (jrawstr `(,digits)))))
       ((fresh (v1 v2 chars1 chars2 result) ; string operations
          (== `(,v1 ,v2) args)
          (typeofo v1 (jstr "string") store~)
          (typeofo v2 (jstr "string") store~)
          (== `(,(jrawstr chars1) ,(jrawstr chars2)) args)
          (conde ((== rator 'string+)
                  (== value (jrawstr result))
                  (appendo chars1 chars2 result))
                 ((== rator 'string<)
                  (string-lesso chars1 chars2 value)))))))))
;; Mutable references: dereferencing (Section 2.2.3)
((fresh (addr-expr addr-val value^)
   (== expr (jderef addr-expr))
   (eval-envo addr-expr env value^ store store~
              next-address next-address~)
   (effect-propagateo value^ value store~ store~
                      next-address~ next-address~
     (== value^ (jref addr-val))
```

```
      (indexo store~ addr-val value))))
  ;; Mutable references: assignment (Section 2.2.3)
  ((fresh (addr-expr addr-val stored-expr stored-val store^ value^)
     (== expr (jassign addr-expr stored-expr))
     (eval-env-listo `(,addr-expr ,stored-expr) env value^ store store^
                     next-address next-address~)
     (effect-propagateo value^ value store^ store~ next-address~ next-address~
       (== value^ (value-list `(,(jref addr-val) ,stored-val)))
       (== value stored-val)
       (set-indexo store^ addr-val stored-val store~))))
  ;; Object field retrieval (2.2.4)
  ((fresh (obj-expr obj-bindings key-expr key-val value^)
     (== expr (jget obj-expr key-expr))
     (eval-env-listo `(,key-expr ,obj-expr) env value^ store store~
                     next-address next-address~)
     (effect-propagateo value^ value store~ store~ next-address~ next-address~
       (== value^ (value-list `(,key-val ,(jobj obj-bindings))))
       (typeofo key-val (jstr "string") store~)
       (conde ((alist-refo key-val obj-bindings value))     ; found
              ((== value (jundef))                          ; not found
               (absent-keyso key-val obj-bindings))))))
  ;; Object field create/update (2.2.4)
  ((fresh (obj-expr obj-bindings obj-bindings^
           key-expr key-val rhs-expr rhs-val value^)
     (== expr (jset obj-expr key-expr rhs-expr))
     (eval-env-listo `(,obj-expr ,key-expr ,rhs-expr) env value^ store store~
                     next-address next-address~)
     (effect-propagateo value^ value store~ store~ next-address~ next-address~
       (== value^ (value-list `(,(jobj obj-bindings) ,key-val ,rhs-val)))
       (== value (jobj obj-bindings^))
       (typeofo key-val (jstr "string") store~)
       (updateo obj-bindings key-val rhs-val obj-bindings^))))
  ;; Function application (Section 2.2.4)
  ((fresh (func params rands args body
           param-arg-bindings
           cenv cenv^ ; closure environment before/after param-arg-bindings
           value^ store^ next-address^)
     (== expr (japp func rands))
     (eval-env-listo `(,func . ,rands) env value^ store store^
                     next-address next-address^)
     (effect-propagateo value^ value store^ store~ next-address^ next-address~
       (== value^ (value-list `(,(jclo params body cenv) . ,args)))
       (zipo params args param-arg-bindings)
       (appendo param-arg-bindings cenv cenv^)
       (eval-envo body cenv^ value store^ store~
                  next-address^ next-address~))))
```

```
;; Throw expressions (Section 2.2.5)
((fresh (label thrown-expr thrown-val) ;; Throw
   (== expr (jthrow label thrown-expr))
   (eval-envo thrown-expr env thrown-val store store~
              next-address next-address~)
   (effect-propagateo thrown-val value store~ store~
                      next-address~ next-address~
     (== value (jbrk label thrown-val)))))
;; Let expressions (Section 2.2.2)
((fresh (lhs-var          ; variable being bound
         rhs-expr rhs-val ; right-hand side expression & value
         store^ next-address^ ; store produced by evaluating the rhs
         body
         let-env)         ; environment after the let binding
   (== expr (jlet lhs-var rhs-expr body))
   (eval-envo rhs-expr env rhs-val store store^ next-address next-address^)
   (effect-propagateo rhs-val value store^ store~
                      next-address^ next-address~
     (== let-env `((,lhs-var . ,rhs-val) . ,env))
     (eval-envo body let-env value store^ store~
                next-address^ next-address~))))
;; Function definition (Section 2.2.4)
((fresh (body params)
   (== expr (jfun params body))
   (== value (jclo params body env))
   (== store store~)
   (== next-address next-address~)))
;; Mutable references: memory allocation (Section 2.2.3)
((fresh (stored-expr stored-val next-address^ store^)
   (== expr (jall stored-expr))
   (eval-envo stored-expr env stored-val store store^
              next-address next-address^)
   (effect-propagateo stored-val value store^ store~
                      next-address^ next-address~
     (== value (jref next-address^))
     (appendo store^ `(,stored-val) store~)
     (incremento next-address^ next-address~))))
;; Object field delete (Section 2.2.4)
((fresh (obj-expr obj-bindings obj-bindings^
         key-expr key-val value^)
   (== expr (jdel obj-expr key-expr))
   (eval-env-listo `(,obj-expr ,key-expr) env value^ store store~
                   next-address next-address~)
   (effect-propagateo value^ value store~ store~ next-address~ next-address~
     (== value^ (value-list `(,(jobj obj-bindings) ,key-val)))
     (== value (jobj obj-bindings^))
```

```
      (typeofo key-val (jstr "string") store~)
      (deleteo obj-bindings key-val obj-bindings^))))
  ;; Begin expression (Section 2.2.5)
  ((fresh (first-expr second-expr first-val value^)
     (== expr (jbeg first-expr second-expr))
     (eval-env-listo `(,first-expr ,second-expr) env value^ store store~
                     next-address next-address~)
     (effect-propagateo value^ value store~ store~ next-address~ next-address~
       (== value^ (value-list `(,first-val ,value))))))
  ;; If expression (Section 2.2.5)
  ((fresh (cond-expr cond-val then-expr else-expr store^ next-address^)
     (== expr (jif cond-expr then-expr else-expr))
     (eval-envo cond-expr env cond-val store store^ next-address next-address^)
     (effect-propagateo cond-val value store^ store~ next-address^ next-address~
       (conde ((== cond-val (jbool #f))
               (eval-envo else-expr env value store^ store~
                          next-address^ next-address~))
              ((== cond-val (jbool #t))
               (eval-envo then-expr env value store^ store~
                          next-address^ next-address~))))))
  ;; While expression (Section 2.2.5)
  ((fresh (cond-expr cond-val body-expr store^ next-address^)
     (== expr (jwhile cond-expr body-expr))
     (eval-envo cond-expr env cond-val store store^
                next-address next-address^)
     (effect-propagateo cond-val value store^ store~
                        next-address^ next-address~
       (conde ((== cond-val (jbool #f))
               (== value (jundef))
               (== store^ store~)
               (== next-address^ next-address~))
              ((== cond-val (jbool #t))
               (eval-envo (jbeg body-expr (jwhile cond-expr body-expr))
                          env value store^ store~
                          next-address^ next-address~))))))
  ;; Try/finally expression (Section 2.2.5)
  ((fresh (try-expr try-val finally-expr finally-val store^ next-address^)
     (== expr (jfin try-expr finally-expr))
     (eval-envo try-expr env try-val store store^ next-address next-address^)
     (eval-envo finally-expr env finally-val store^ store~
                next-address^ next-address~)
     (effect-propagateo finally-val value store~ store~
                        next-address~ next-address~
       (== value try-val))))
  ;; Try/catch expression (Section 2.2.5)
  ((fresh (try-expr try-val try-val-tag try-val-payload
```

```
            catch-label catch-var catch-expr
            break-label break-val
            store^ next-address^ env^)
       (== expr (jcatch catch-label try-expr catch-var catch-expr))
       (eval-envo try-expr env try-val store store^ next-address next-address^)
       (conde ((== try-val (jbrk break-label break-val)) ; break doesn't match
               (== `(,value ,store~ ,next-address~)
                   `(,try-val ,store^ ,next-address^))
               (=/= break-label catch-label))
              ((== try-val `(,try-val-tag . ,try-val-payload)) ; no break caught
               (== `(,value ,store~ ,next-address~)
                   `(,try-val ,store^ ,next-address^))
               (=/= try-val-tag 'break))
              ((== try-val (jbrk catch-label break-val)) ; break caught
               (appendo `((,catch-var . ,break-val)) env env^)
               (eval-envo catch-expr env^ value store^ store~
                          next-address^ next-address~)))))))

;; Sequentially evaluate a list of LambdaJS expressions
(define (eval-env-listo exprs env vals store store~ next-address next-address~)
  (conde
    ; base case: empty list of expressions
    ((== exprs '())
     (== vals (value-list '()))
     (== store store~)
     (== next-address next-address~))
    ; non-empty list of expressions
    ((fresh (expr expr-rest val val-rest val-rest-payload store^ next-address^)
       (== exprs `(,expr . ,expr-rest))
       (eval-envo expr env val store store^ next-address next-address^)
       (effect-propagateo val vals store^ store~ next-address^ next-address~
         (eval-env-listo expr-rest env val-rest store^ store~
                         next-address^ next-address~)
         (effect-propagateo val-rest vals store~ store~
                            next-address~ next-address~
           (== val-rest (value-list val-rest-payload))
           (== vals (value-list `(,val . ,val-rest-payload)))))))))

; For propagating control effects (Section 2.2.6)
(define-syntax-rule (effect-propagateo value^        value~
                                       store^        store~
                                       next-address^ next-address~
                                       cont ...)
  (fresh (label bval tag rest)
    (conde ((== (jbrk label bval) value^)
            (== `(,value~ ,store~ ,next-address~)
```

```
              `(,value^ ,store^ ,next-address^)))
           ((== `(,tag . ,rest) value^)
            (=/= tag 'break)
            cont ...))))

(define (typeofo value type store)
  (fresh (temp)
    (conde ((== value (jundef))
            (== type (jstr "undefined")))
           ((== value (jnul))
            (== type (jstr "object")))
           ((== value `(string . ,temp))
            (== type (jstr "string")))
           ((== value  `(number . ,temp))
            (== type (jstr "number")))
           ((== value  `(boolean . ,temp))
            (== type (jstr "boolean")))
           ((fresh (fields priv call)
              (== value (jref temp))
              (conde ((== type (jstr "object"))
                      (indexo store temp `(object ,fields))
                      (lookupo (jstr "private") fields (jobj priv))
                      (absent-keyso (jstr "call") priv))
                     ((== type (jstr "function"))
                      (indexo store temp `(object ,fields))
                      (lookupo (jstr "private") fields (jobj priv))
                      (lookupo (jstr "call") priv call))))))))

(define (string-lesso s1 s2 value)
  (fresh (x x^ y y^ rest)
    (conde ((== s1 '())
            (== s2 `(,x . ,rest))
            (== value (jbool #t)))
           ((== s2 '())
            (== value (jbool #f)))
           ((== s1 `(,x . ,x^))
            (== s2 `(,x . ,y^))
            (string-lesso x^ y^ value))
           ((== s1 `(,x . ,x^))
            (== s2 `(,y . ,y^))
            (== value (jbool #t))
            (=/= x y)
            (<o x y))
           ((== s1 `(,x . ,x^))
            (== s2 `(,y . ,y^))
            (== value (jbool #f))
```

```
            (=/= x y)
            (<=o y x)))))

(define (value-list values)
  (cons 'value-list values))
```

B RELATIONAL S-EXPRESSION JAVASCRIPT TO LAMBDAJS PARSER

```
; Parse a JavaScript statement with human-readable literals
(define (parseo/readable stmt jexpr)
  (parse-topo (dehumanize stmt) jexpr))

; Parse a JavaScript statement with relational number and string literals
(define (parse-topo stmt jexpr)
  (fresh (vars exp^ allocations body)
    (hoist-varo stmt vars)
    (allocateo vars body jexpr)
    (parseo stmt body)))

; Parse a JavaScript statement to LambdaJS expression
(define (parseo stmt jexpr)
  (conde
    ; variable declaration (Section 3.2.1)
    ((fresh (vars)
       (== stmt `(var . ,vars))
       (parse-var-assignmentso vars jexpr)))
    ; expressions have a helper for their own
    ((parse-expro stmt jexpr))
    ; begin statement (Section 3.2.7)
    ((fresh (stmts jexprs begin-jexpr)
       (== stmt `(begin . ,stmts))
       (== jexpr (jbeg begin-jexpr (jundef)))
       (parse-listo stmts jexprs)
       (begino jexprs begin-jexpr)))
    ; if statement  (Section 3.2.7)
    ((fresh (cond-expr then-stmt else-stmt cond-jexpr then-jexpr else-jexpr)
       (== stmt `(if ,cond-expr ,then-stmt ,else-stmt))
       (== jexpr (jbeg (jif cond-jexpr then-jexpr else-jexpr) (jundef)))
       (parse-expro cond-expr cond-jexpr)
       (parse-listo `(,then-stmt ,else-stmt) `(,then-jexpr ,else-jexpr))))
    ; for loops (Section 3.2.7)
    ((fresh (init-stmt init-jexpr
             cond-expr cond-jexpr
             inc-expr inc-jexpr
             body-stmts body-jexprs body-jexpr)
       (== stmt `(for (,init-stmt ,cond-expr ,inc-expr) . ,body-stmts))
```

```
   (== jexpr (jbeg init-jexpr
                  (jcatch 'break
                          (jwhile cond-jexpr (jbeg body-jexpr inc-jexpr))
                          'e (jundef))))
   (parse-expr-listo `(,cond-expr ,inc-expr) `(,cond-jexpr ,inc-jexpr))
   (parseo init-stmt init-jexpr)
   (parse-listo body-stmts body-jexprs)
   (begino body-jexprs body-jexpr)))
; return control effect (Section 3.2.8)
((fresh (val-expr val-jexpr)
   (== stmt `(return ,val-expr))
   (== jexpr (jthrow 'return val-jexpr))
   (parse-expro val-expr val-jexpr)))
; throw control effect (Section 3.2.8)
((fresh (val-expr val-jexpr)  ;; throw
   (== stmt `(throw ,val-expr))
   (== jexpr (jthrow 'error val-jexpr))
   (parse-expro val-expr val-jexpr)))
; break control effect (Section 3.2.8)
((== stmt `(break)) (== jexpr (jthrow 'break (jundef))))
; try/catch control effect (Section 3.2.8)
((fresh (try-stmt try-jexpr catch-stmt catch-jexpr catch-var)
   (== stmt `(try ,try-stmt catch ,catch-var ,catch-stmt))
   (== jexpr
       (jbeg (jcatch 'error try-jexpr catch-var
                     (jlet catch-var (jall (jvar catch-var)) catch-jexpr))
             (jundef)))
   (parseo try-stmt try-jexpr)
   (parseo catch-stmt catch-jexpr)))
; try/finally control effect (Section 3.2.8)
((fresh (try-stmt try-jexpr finally-stmt finally-jexpr)
   (== stmt `(try ,try-stmt finally ,finally-stmt))
   (== jexpr (jfin try-jexpr (jbeg finally-jexpr (jundef))))
   (parseo try-stmt try-jexpr)
   (parseo finally-stmt finally-jexpr)))
; try/catch/finally control effect (Section 3.2.8)
((fresh (try-stmt try-jexpr
         catch-stmt catch-jexpr catch-var
         finally-stmt finally-jexpr)
   (== stmt `(try ,try-stmt
              catch ,catch-var ,catch-stmt
              finally ,finally-stmt))
   (== jexpr (jbeg (jcatch 'error try-jexpr catch-var
                           (jlet catch-var (jall (jvar catch-var))
                                 catch-jexpr))
                   (jbeg finally-jexpr (jundef))))
```

```
       (parseo try-stmt try-jexpr)
       (parseo catch-stmt catch-jexpr)
       (parseo finally-stmt finally-jexpr)))
    ; while statements (Section 3.2.7)
    ((fresh (cond-expr cond-jexpr body-stmts body-jexprs body-jexpr)
       (== stmt `(while ,cond-expr . ,body-stmts))
       (== jexpr (jcatch 'break (jwhile cond-jexpr body-jexpr) 'e (jundef)))
       (parse-expro cond-expr cond-jexpr)
       (parse-listo body-stmts body-jexprs)
       (begino body-jexprs body-jexpr)))))

; Parse a JavaScript expression to LambdaJS expression
(define (parse-expro expr jexpr)
  (conde
    ;; variables (Section 3.2.1)
    ((symbolo expr) (== jexpr (jderef (jvar expr))))
    ;; simple literals
    ((conde ((== expr jexpr) (conde ((fresh (x) (== expr (jrawnum x))))
                                    ((fresh (x) (== expr (jrawstr x))))))
            ((== expr #t) (== jexpr (jbool #t)))
            ((== expr #f) (== jexpr (jbool #f)))
            ((== expr (jnul)) (== jexpr (jnul)))
            ((== expr (jundef)) (== jexpr (jundef)))))
    ;; builtin operations (Section 3.2.5)
    ((fresh (rator rands rand-jexprs)
       (== expr `(op ,rator . ,rands))
       (== jexpr (jdelta rator rand-jexprs))
       (parse-expr-listo rands rand-jexprs)))
    ;; assignments (Section 3.2.6)
    ((fresh (lhs-expr rhs-expr rhs-jexpr)
       (== expr `(:= ,lhs-expr ,rhs-expr))
       (conde ((symbolo lhs-expr)
               (== jexpr (jassign (jvar lhs-expr) rhs-jexpr))
               (parse-expro rhs-expr rhs-jexpr))
              ((fresh (obj-expr obj-jexpr key-expr key-jexpr)
                 (== lhs-expr `(@ ,obj-expr ,key-expr))
                 (== jexpr
                     (japp (jfun '(obj key rhs)
                                 (jbeg (jassign
                                         (jvar 'obj)
                                         (jset (jderef (jvar 'obj))
                                               (jstr "public")
                                               (jset (jget (jderef (jvar 'obj))
                                                           (jstr "public"))
                                                     (jvar 'key) (jvar 'rhs))))
                                       (jvar 'rhs)))
```

```
                        (list obj-jexpr key-jexpr rhs-jexpr)))
             (parse-expro obj-expr obj-jexpr)
             (parse-expro key-expr key-jexpr)
             (parse-expro rhs-expr rhs-jexpr))))))
;; object field access (Section 3.2.2)
((fresh (obj-expr obj-jexpr field-expr field-jexpr)
   (== expr `(@ ,obj-expr ,field-expr))
   (== jexpr (jget (jget (jderef obj-jexpr) (jstr "public")) field-jexpr))
   (parse-expro obj-expr obj-jexpr)
   (parse-expro field-expr field-jexpr)))
;; Function call (Section 3.2.4)
((fresh (func-expr func-jexpr arg-exprs arg-jexprs)
   (== expr `(call ,func-expr . ,arg-exprs))
   (== jexpr (japp (jget (jget (jderef func-jexpr) (jstr "private"))
                         (jstr "call"))
                   arg-jexprs))
   (parse-expro func-expr func-jexpr)
   (parse-expr-listo arg-exprs arg-jexprs)))
;; object creation (Section 3.2.2)
((fresh (binding-exprs public-jexpr)
   (== expr `(object . ,binding-exprs))
   (== jexpr (jall (jset (jobj `((,(jstr "private") . ,(jobj '()))))
                         (jstr "public") public-jexpr)))
   (parse-obj-bindingso binding-exprs public-jexpr)))
;; function definition (Section 3.2.3)
((fresh (params
          body-stmts ; javascript statements
          body-jexprs ; a list of LambdaJS expressions
          body-jexpr ; a single LambdaJS begin expression
          body-jexpr/vars
          body-jexpr/vars+params
          vars
          hoisted-vars
          return-var)
   (== expr `(function ,params . ,body-stmts))
   (== jexpr (jall (jset (jobj `((,(jstr "public") . ,(jobj '()))))
                         (jstr "private")
                         (jset (jobj '()) (jstr "call")
                               (jfun params
                                     (jcatch 'return
                                             (jbeg body-jexpr/vars+params
                                                   (jundef))
                                             'result
                                             (jvar 'result)))))))
   (hoist-var-listo body-stmts vars)
   (differenceo vars params hoisted-vars)
```

```
       (parse-listo body-stmts body-jexprs)
       (begino body-jexprs body-jexpr)
       (allocateo hoisted-vars body-jexpr body-jexpr/vars)
       (assigno params body-jexpr/vars body-jexpr/vars+params)))
    ;; Comma expression sequencing (not in paper since we can't decide on a name)
    ((fresh (exps exps^)
       (== expr `(comma . ,exps))
       (parse-expr-listo exps exps^)
       (begino exps^ jexpr)))))

; Parse a list of statements
(define (parse-listo stmts jexprs)
  (conde ((== stmts '()) (== jexprs '()))
         ((fresh (stmt jexpr stmts-rest jexprs-rest)
            (== stmts `(,stmt . ,stmts-rest))
            (== jexprs `(,jexpr . ,jexprs-rest))
            (parseo stmt jexpr)
            (parse-listo stmts-rest jexprs-rest)))))

; Parse a list of expressions
(define (parse-expr-listo exprs jexprs)
  (conde ((== exprs '()) (== jexprs '()))
         ((fresh (expr jexpr exprs-rest jexprs-rest)
            (== exprs `(,expr . ,exprs-rest))
            (== jexprs `(,jexpr . ,jexprs-rest))
            (parse-expro expr jexpr)
            (parse-expr-listo exprs-rest jexprs-rest)))))

(define (parse-var-assignmentso vars assignments-jexpr)
  (conde ((== vars '()) (== assignments-jexpr (jundef)))
         ((fresh (var-name var-expr var-jexpr vars-rest assignments-rest-jexpr)
            (== vars `((,var-name ,var-expr) . ,vars-rest))
            (== assignments-jexpr (jbeg (jassign (jvar var-name) var-jexpr)
                                        assignments-rest-jexpr))
            (parse-expro var-expr var-jexpr)
            (parse-var-assignmentso vars-rest assignments-rest-jexpr)))
         ((fresh (var-name vars-rest)
            (== vars `(,var-name . ,vars-rest))
            (symbolo var-name)
            (parse-var-assignmentso vars-rest assignments-jexpr)))))

; object {...}
(define (parse-obj-bindingso binding-exprs obj-jexpr)
  (conde ((== binding-exprs '()) (== obj-jexpr (jobj '())))
         ((fresh (field-expr field-jexpr val-expr val-jexpr
                  binding-exprs-rest obj-jexpr-rest)
```

```
            (== binding-exprs  `((,field-expr ,val-expr) . ,binding-exprs-rest))
            (== obj-jexpr (jset obj-jexpr-rest field-jexpr val-jexpr))
            (parse-expro field-expr field-jexpr)
            (parse-expro val-expr val-jexpr)
            (parse-obj-bindingso binding-exprs-rest obj-jexpr-rest)))))

; Build nested LambdaJS begin out of a list of LambdaJS exprs
(define (begino jexprs jexpr)
  (conde ((== jexprs '()) (== jexpr (jundef)))
         ((== jexprs `(,jexpr)))
         ((fresh (first rest rest-jexpr)
            (== jexprs`(,first . ,rest))
            (=/= rest '())
            (== jexpr (jbeg first rest-jexpr))
            (begino rest rest-jexpr)))))

; Hoist declared variables out of a statement
(define (hoist-varo stmt vars)
  (conde ((fresh (x) (== stmt `(var . ,x)) (var-nameso x vars)))
         ((== vars '())  ;; These never embed var declarations
          (conde ((fresh (x) (== stmt `(return ,x))))
                 ((fresh (x) (== stmt `(throw ,x))))
                 ((fresh (x) (== stmt `(number ,x))))
                 ((fresh (x) (== stmt `(string ,x))))
                 ((fresh (x) (== stmt `(object . ,x))))
                 ((fresh (x) (== stmt `(comma . ,x))))
                 ((== stmt #t))
                 ((== stmt #f))
                 ((== stmt (jundef)))
                 ((== stmt (jnul)))
                 ((symbolo stmt))
                 ((fresh (func args) (== stmt `(call ,func . ,args))))
                 ((fresh (erest) (== stmt `(function . ,erest))))
                 ((== stmt `(break)))
                 ((fresh (x) (== stmt `(op . ,x))))
                 ((fresh (x) (== stmt `(@ . ,x))))
                 ((fresh (x) (== stmt `(:= . ,x))))))
         ((fresh (try-stmt catch-stmt catch-var)
            (== stmt `(try ,try-stmt catch ,catch-var ,catch-stmt))
            (hoist-var-listo `(,try-stmt ,catch-stmt) vars)))
         ((fresh (try-stmt catch-stmt finally-stmt catch-var)
            (== stmt `(try ,try-stmt catch ,catch-var ,catch-stmt
                           finally ,finally-stmt))
            (hoist-var-listo `(,try-stmt ,catch-stmt ,finally-stmt) vars)))
         ((fresh (cond then else)
            (== stmt `(if ,cond ,then ,else))
```

```
           (hoist-var-listo `(,then ,else) vars)))
         ((fresh (cond body)
           (== stmt `(while ,cond . ,body))
           (hoist-var-listo body vars)))
         ((fresh (exps)
           (== stmt `(begin . ,exps))
           (hoist-var-listo exps vars)))
         ((fresh (init cond inc body)
           (== stmt `(for (,init ,cond ,inc) . ,body))
           (hoist-var-listo `(,init . ,body) vars)))))

(define (hoist-var-listo stmts vars)
  (conde ((== stmts '()) (== vars '()))
         ((fresh (stmt stmts-rest vars-first vars-rest)
           (== stmts `(,stmt . ,stmts-rest))
           (hoist-varo stmt vars-first)
           (hoist-var-listo stmts-rest vars-rest)
           (appendo vars-first vars-rest vars)))))

(define (var-nameso var-decls names)
  (conde ((== var-decls '()) (== names '()))
         ((fresh (var-decl var-decls-rest name expr names-rest)
           (== var-decls `(,var-decl . ,var-decls-rest))
           (== names `(,name . ,names-rest))
           (conde ((== var-decl `(,name ,expr)))
                  ((symbolo var-decl) (== name var-decl)))
           (var-nameso var-decls-rest names-rest)))))

(define (allocateo var-names body-jexpr full-jexpr)
  (conde ((== var-names '()) (== full-jexpr body-jexpr))
         ((fresh (var-name vars-rest rest-jexpr)
           (== var-names `(,var-name . ,vars-rest))
           (== full-jexpr (jlet var-name (jall (jundef)) rest-jexpr))
           (allocateo vars-rest body-jexpr rest-jexpr)))))

(define (assigno var-names body-jexpr full-jexpr)
  (conde ((== var-names '()) (== full-jexpr body-jexpr))
         ((fresh (var-name vars-rest rest-jexpr)
           (== var-names `(,var-name . ,vars-rest))
           (== full-jexpr (jlet var-name (jall (jvar var-name)) rest-jexpr))
           (assigno vars-rest body-jexpr rest-jexpr)))))

; list (set) difference operation
(define (differenceo items toremove remaining)
  (conde ((== items '()) (== remaining items))
         ((fresh (el rest remaining-rest)
```

```
            (== items `(,el . ,rest))
            (conde ((== remaining `(,el . ,remaining-rest))
                    (not-in-listo el toremove)
                    (differenceo rest toremove remaining-rest))
                   ((membero el toremove)
                    (differenceo rest toremove remaining)))))))

(define (humanize-string x)
  (define (humanize-char x)
    (define n (mknum->num x))
    (if (integer? n) (integer->char n) n))
  (if (list? x)
    (let ((cs (map humanize-char x)))
      (if (andmap char? cs) (list->string cs) `(string ,cs)))
    `(string ,x)))

(define/match (humanize stmt)
  [((list 'string x)) (humanize-string x)]
  [((list 'number x))
   (define result (mknum->num x))
   (if (integer? result) result `(number ,result))]
  [((list)) '()]
  [((? list?)) (cons (humanize (car stmt)) (humanize (cdr stmt)))]
  [(_) stmt])

(define/match (dehumanize stmt)
  [((? string?)) (jstr stmt)]
  [((? integer?)) (jnum stmt)]
  [((list)) '()]
  [((? list?)) (cons (dehumanize (car stmt)) (dehumanize (cdr stmt)))]
  [(_) stmt])

(define (mknum->num xs)
  (if (and (list? xs) (andmap integer? xs))
    (foldr (lambda (d n) (+ d (* 2 n))) 0 xs)
    xs))
```

C LAMBDAJS STRUCTURE DEFINITIONS

```
(define (jlet key value exp)
  `(let ,key ,value ,exp))

(define (jfun params body)
  `(fun ,params ,body))

(define (jclo params body env)
```

```
  `(jclosure ,params ,body ,env))

(define (japp closure args)
  `(app ,closure ,args))

(define (jget obj key)
  `(get ,obj ,key))

(define (jset obj key value)
  `(set ,obj ,key ,value))

(define (jdel obj key)
  `(delete ,obj ,key))

(define (jvar var)
  `(var ,var))

(define (jrawnum n)
  `(number ,n))

(define (jnum n)
  `(number ,(build-num n)))

(define (jobj bindings)
  `(object ,bindings))

(define (jall value)
  `(allocate ,value))

(define (jref value)
  `(ref ,value))

(define (jderef address)
  `(deref ,address))

(define (jassign var val)
  `(assign ,var ,val))

(define (jbeg first second)
  `(begin ,first ,second))

(define (jbool bool)
  `(boolean ,bool))

(define (jif cond then else)
  `(if ,cond ,then ,else))
```

```
(define (jundef)
  '(undefined))

(define (jnul)
  '(null))

(define (jwhile cond body)
  `(while ,cond ,body))

(define (jthrow label value)
  `(throw ,label ,value))

(define (jbrk label value)
  `(break ,label ,value))

(define (jfin try-exp fin-exp)
  `(finally ,try-exp ,fin-exp))

(define (jcatch label try-exp catch-var catch-exp)
  `(catch ,label ,try-exp ,catch-var ,catch-exp))

(define (jrawstr str)
  `(string ,str))

(define (jstr str)
  `(string ,(map (compose1 build-num char->integer) (string->list str))))

(define (jdelta fun vals)
  `(delta ,fun ,vals))

(define (jpass) 'pass)
```

D OTHER RELATIONAL UTILITIES

```
(define (incremento in out)
  (== out `(,in)))

(define (decremento in out)
  (incremento out in))

(define (absent-keyso key obj)
  (conde ((== obj '()))
         ((fresh (k v rest)
            (== obj `((,k . ,v) . ,rest))
            (=/= k key)
```

```
            (absent-keyso key rest)))))

(define (updateo obj key value result)
  (conde ((== obj '())
          (== result `((,key . ,value))))
         ((fresh (orest rrest k v v2)
            (== obj    `((,k . ,v ) . ,orest))
            (== result `((,k . ,v2) . ,rrest))
            (conde ((== k key)
                    (== v2 value)
                    (== orest rrest))
                   ((=/= k key)
                    (== v2 v)
                    (updateo orest key value rrest)))))))

(define (deleteo obj key result)
  (conde ((== obj '()) (== result '()))
         ((fresh (orest rrest k v)
            (== obj `((,k . ,v) . ,orest))
            (conde ((== k key)
                    (== orest result))
                   ((== result `((,k . ,v) . ,rrest))
                    (=/= k key)
                    (deleteo orest key rrest)))))))

(define (appendo s t r)
  (conde ((== s '()) (== t r))
         ((fresh (r^ sel srest)
            (== r `(,sel . ,r^))
            (== s `(,sel . ,srest))
            (appendo srest t r^)))))

(define (zipo as ds pairs)
  (conde ((== as '()) (== ds '()) (== pairs '()))
         ((fresh (a as-rest d ds-rest pairs-rest)
            (== as `(,a . ,as-rest))
            (== ds `(,d . ,ds-rest))
            (== pairs `((,a . ,d) . ,pairs-rest))
            (zipo as-rest ds-rest pairs-rest)))))

(define (not-in-listo el list)
  (conde
    ((== list '()))
    ((fresh (x rest)
       (== `(,x . ,rest) list)
       (=/= el x)
```

```
       (not-in-listo el rest)))))

(define (lookupo x env t)
  (fresh (rest y v)
    (== `((,y . ,v) . ,rest) env)
    (conde
      ((== y x) (== v t))
      ((=/= y x) (lookupo x rest t)))))

(define (indexo lst index result)
  (fresh (l lrest index^)
    (== lst `(,l . ,lrest))
    (conde ((== index '())
            (== result l))
           ((=/= index '())
            (decremento index index^)
            (indexo lrest index^ result)))))

(define (set-indexo lst index value result)
  (fresh (l lrest rrest index^)
    (== lst `(,l . ,lrest))
    (conde ((== index '())
            (== result `(,value . ,lrest)))
           ((== result `(,l . ,rrest))
            (=/= index '())
            (decremento index index^)
            (set-indexo lrest index^ value rrest)))))

(define (membero item lst)
  (fresh (first rest)
    (== lst `(,first . ,rest))
    (conde ((== first item))
           ((=/= first item) (membero item rest)))))

(define (no-alist-refo key lst)
  (conde ((== '() lst))
         ((fresh (first-key first-value rest)
            (== lst `((,first-key . ,first-value) . ,rest))
            (=/= key first-key)
            (no-alist-refo key rest)))))

(define (alist-refo key lst value)
  (fresh (first-key first-value rest)
    (== lst `((,first-key . ,first-value) . ,rest))
    (conde ((== first-key key)
            (== value first-value)
```

```
 (no-alist-refo key rest))
((=/= first-key key)
 (alist-refo key rest value)))))
```